Using OS Observations to Improve Performance in Multicore Systems

Today's operating systems don't adequately handle the complexities of multicore processors. Architectural features confound existing OS techniques for task scheduling, load balancing, and power management. This article shows that the OS can use data obtained from dynamic runtime observation of task behavior to ameliorate performance variability and more effectively exploit multicore processor resources. The authors' research prototypes demonstrate the utility of observation-based policy.

Rob Knauerhase
Paul Brett
Barbara Hohlt
Tong Li
Scott Hahn
Intel

Multicore processors include several independent processing units in a single package.[1] These chip multiprocessing (CMP) processors are already available in dual- and quad-core desktop and server platforms.[2] The cores in a CMP system can share certain resources—for example, last-level caches or various internal system buses. Another type of multicore system is simultaneous multithreaded (SMT) processors, also known as hyperthreaded processors.[3] These chips look like multiple cores, but they implement each virtual core with a combination of functional units within a traditional processor. By their nature, SMT processors share resources within a die, but they achieve good parallelism when application threads don't compete among themselves.

The roadmaps of most major microprocessor manufacturers include proliferations of multicore designs—for example, designs containing many hyperthreaded cores. It's not unreasonable to expect the complexity of such systems to increase over time. Indeed, Intel has demonstrated a prototype system with 80 cores.[4]

In general, operating systems exploit multicore systems by using a multiprocessing kernel (Linux and Microsoft Windows use a symmetric-multiprocessing, or SMP, kernel). The kernel treats each core as though it were a completely separate processor, with separate caches sharing a coherent view of main memory. Applications today are written to expose software execution threads, and the OS maps these threads onto computation resources (SMP processors). Although the SMP kernel also works for CMP systems, it is naive with respect to the processor's actual internal workings—it cannot exploit CMP features and cannot avoid CMP challenges.

Modern processors also include hardware features for monitoring the CPU's perfor-

Published by the IEEE Computer Society

IEEE Micro micr-28-03-knau.3d 28/5/08 14:44:49 54 Cust # Knauerhase

mance and behavior. Programmers often use these counters to improve application performance by monitoring sections of code to detect and optimize hotspots in a program's execution. For example, the Intel Vtune system allows programmers to optimize for cache usage and floating-point and multimedia extensions (MMX) instruction usage.

We posit that in a multicore environment, the OS can and should make observations of the behavior of threads running in the system. These observations, combined with knowledge of the processor architecture, allow the implementation of different policies in the OS. Good policies can improve overall system performance, improve application performance, decrease system power consumption, or provide arbitrary, user-defined combinations of these benefits. The data we present here, from different processor and system configurations and two different operating systems, support our hypotheses.

Experimental environments

Our experimentation involved several different software and hardware environments. In this section, we describe the most representative configurations, some of the challenges these environments present, and the prototypes we built to validate our observation-based scheduling policies.

Software environment

For quite some time, application programming has included the notion of execution threads. Programming languages and libraries support mechanisms to spawn threads, to communicate between threads, and to synchronize execution among threads. Operating systems use a scheduler to multiplex threads on a unicore processor or map them to a set of SMP processors. At compile time, programmers usually have no knowledge of the configuration of the machine that will execute the application (other than its instruction set architecture). At runtime, applications—or system administrators—can only express basic hints about preferred options to the OS by assigning relative priorities and processor affinities to certain threads.

Contemporary operating systems attempt to schedule execution threads in a fair and low-overhead manner. Most simply, they balance the load across processors by migrating tasks to keep the run queues approximately equal. Some optimizations to this scheme are also possible; for example, recent Linux kernels hesitate to migrate a thread to a different SMP processor if the cache is "hot" (that is, a large quantity of the thread's data appear to be present in the cache).

Our work involves modifying OS decisions to explicitly accommodate multicore processors under dynamic workloads.

Challenges

The platforms we have studied present several difficulties:

- *Cache interference in the last-level cache (LLC)*. If a task runs on core A, it can use the entire LLC. Another task, running on core B, shares the LLC resource; the resulting contention slows both tasks. Worse, the amount of contention is quite dynamic because it depends on each task's behavior at a given time. This behavior is impossible for the application to know at compile time.
- *Lack of intelligent thread migration*. Linux and OS X include migration for basic load balancing, but they treat each core equivalently, without the notion of resources shared among cores.
- *No accommodation of cores with different features*. Current Intel-compatible multicore implementations feature cores that are exact copies of each other. In the future, however, some cores will likely be functionally asymmetric for reasons of power, die area, cost, and complexity.

We have explored three policies in detail to address these challenges. Because cache interference can severely degrade task performance, we developed policies that use observed task behavior to mitigate interference by altering OS scheduler decisions. To keep cache loads approximately equal across

LLCs, we developed a prototype policy that affects OS migration decisions on the basis of similar observations. We investigated mechanisms to improve fairness in cache usage. Finally, to handle cases in which cores provide different features, we implemented a policy of observation-based migration among functionally asymmetric cores. Our results show the benefits of these policies.

Observations let us know about a task without having to know exactly what it is doing. History and hysteresis let us accommodate changes in behavior from phases of an application and adjust accordingly. We have developed an observation subsystem that collects historical and hysteretic data and makes them available to various policies.

System environment

We achieved the Linux results we describe in this article by running kernel version 2.6.20 on a quad-core Intel Xeon 5300 series processor. This processor features two 4-Mbyte LLC arrays, each shared by two cores. We call one cache and the cores that access that cache an *LLC group*.

We performed our experiments on the Macintosh platform under the Macintosh OS X Tiger release on an Intel Core 2 Duo T5600 processor. (Lacking source code to the production operating system, we used the open-source Darwin OS X code base for our Mac experiments.) The chip includes two cores that share a 2-Mbyte LLC.

Prototype systems

We implemented several prototypes to develop our observation-based policies and to demonstrate the benefit derived from each. The foundation of each prototype we describe in this article is our observation subsystem. Atop that, we developed prototype systems to reduce cache interference, to migrate tasks more intelligently, to improve per-task fairness, and to accommodate functional asymmetry within a system. The remainder of the article discusses each of these prototypes and the data we derived from experiments on each.

Observation subsystem

OBS-M inspects relevant performance-monitoring counters and kernel data structures and gathers information on a per-thread basis. By taking a brief snapshot at context-switching time (that is, the beginning and end of each thread's execution period), OBS-M gathers per-thread information in a very low-overhead manner. It then makes the observed data available as input to various policies. Empirical measurements of tasks running with and without OBS-M show that the overhead we add (generally less than 1,000 clock cycles) is virtually indistinguishable from the standard kernel.

For cache-related policies, we program the processor counters[5] for several measurable events:

- LLC misses (INVALID_L2_RQSTS),
- LLC references (L2_RQSTS),
- instructions retired (INSTR_RETIRED.ANY),
- core cycles (CPU_CLK_UNHALTED.CORE), and
- reference cycles (CPU_CLK_UNHALTED.REF).

For feature-related policies, such as those that accommodate functional asymmetry, OBS-M hooks the system exception handler and looks for (among other things) system faults that indicate attempts to use features that the core doesn't offer. When such a fault occurs, OBS-M disassembles the code that caused the fault and notes which unsupported execution was executed. It maintains the counts, types, and frequencies of such illegal instructions on a per-task basis.

Policy: Reducing cache interference

We implement polices for local-cache scheduling in a module called OBS-L. The prototype implementing this policy consumes per-task observation data related to cache usage and affects scheduling decisions to reduce interference among processes running on cores that share an LLC. OBS-L synthesizes a cache weight for each task, using raw metrics from OBS-M.

IEEE Micro micr-28-03-knau.3d 28/5/08 14:45:13 56 Cust # Knauerhase

We explored various definitions of cache weight to find the most useful input to scheduling policy. To gauge cache usage, we considered several choices. For example, the LLC miss ratio distinguishes threads that are likely to sully the cache of their corunners. However, using only the ratio doesn't distinguish between threads with potentially large differences in the absolute number of cache references. We also considered using the absolute number of cache hits, misses, or references. Although these reflect actual cache usage, scaling them (both because of thread behavior and varying quantum length) is a difficult problem. In the end, our experimentation found that cache misses per cycle are the best indication of cache interference.

Another question was how to predict future behavior from historical behavior. After testing our policies with task lifetime averages, running averages, and weighted averages, we decided that simply basing the weight on the metrics from the immediately past quantum (that is, using temporal locality as our predictor) is the best solution. Because the metering system doesn't have detailed knowledge of what a task is doing, it is difficult to detect behavior phases or to predict cache usage burstiness. We expect that higher-level hints in the form of annotations from the compiler or programmer might change this determination, but that is a subject of future work.

We implemented observation and scheduler modifications in Linux and Macintosh OS X, adding a policy to reduce cache interference among cores sharing a last-level data cache. We augmented the run-queue structures in both operating systems to include bins of similarly weighted tasks at the same priority level. The scheduler policy attempts to keep the sum of the cache weights of all the currently running processes close to a medium value. When core 0 is ready to be assigned a new task, the scheduler examines the weights of tasks on other cores and chooses a task whose weight best complements the corunning tasks for core 0. Thus, heavy and light tasks tend to be coscheduled on the shared cache, avoiding the interference that results from coscheduling two heavy tasks. Figure 1 pictorially shows the difference between coscheduling two heavy tasks versus coscheduling heavy and light tasks together in a dual-core processor.

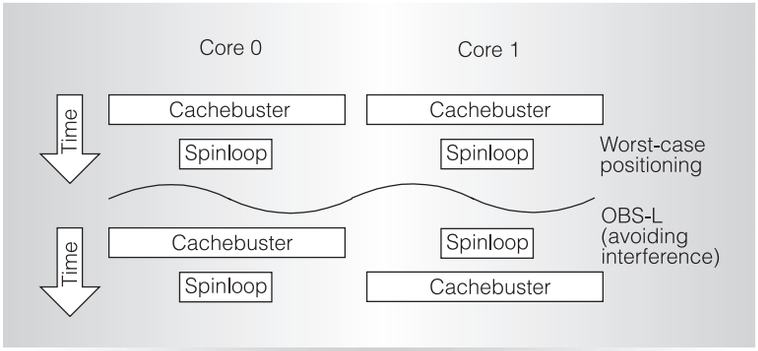

Figure 1. Two ways of coscheduling cache-heavy and cache-light tasks.

Our system preserves task priorities explicitly, so that we don't risk changing fairness or causing starvation among tasks in the OS. In future work, we might relax priorities somewhat to achieve even better overall system performance.

Linux prototype results

First, we ran experiments with microbenchmarks to approximate best-case results. The cachebuster (cb) application simply consumes as much cache as possible as quickly as possible; the algorithm deliberately modifies memory to thwart the LLC's benefit. Spinloop (sl), on the other hand, consumes CPU with a minimum of memory access. For two cores that share an LLC, pairing [cb, sl][cb, sl] results in the worst performance because both cachebuster applications contend for cache resources at the same time. (The notation [x, y][z, w] indicates threads x and y scheduled on one core in order x then y, and threads z and w scheduled on another core in order z then w.) Our experiments showed that in this worst case, cachebuster performance degrades between 26 and 30 percent compared with running in an uncontended environment.

As mentioned earlier, OBS-L attempts to schedule heavy and light tasks together, ideally resulting in a task-to-core mapping of [cb, sl][sl, cb]. OBS-L's observation-based policy often can schedule this way and indeed improves performance, resulting in

IEEE Micro micr-28-03-knau.3d 28/5/08 14:45:13 57 Cust # Knauerhase

an average 30 percent speedup over the normal Linux scheduler (and a 73 percent improvement over the worst-case mapping).

To approximate real-world workloads, we ran our OBS-L experimental setup with a set of applications from the SPEC CPU 2000 suite run continuously in an infinite loop in a method similar to that of Bulpin and Pratt.[6] We based our selection of applications largely on analyses by Zhao et al. to provide a mixture of cache behaviors.[7]

We ran two instances each of mcf, eon, sixtrack, and swim across two cores that share an LLC. The cache-heavy task mcf improved between 18 and 22 percent, and the cache-medium task swim improved 11 percent. However, the lighter-weight tasks suffered a performance degradation of between 1 and 5 percent. To determine overall system performance improvement, we calculated the geometric mean of all tasks' speedup ratios, a method proposed by Zhang et al.[8] The observation-enhanced local scheduler resulted in a net 6 percent overall system speedup.

Figure 2 shows the results for our comparison workload. Each group along the horizontal axis represents one application in the workload; the group labeled ''Overall'' shows combined results for the whole system. The ''Stock'' bars indicate the performance under unmodified Linux; this is our normalized value, and as such is always 1.00.

Macintosh prototype results

We ran similar experiments on our Macintosh prototype. Tests with cachebuster and spinlock demonstrated its benefit in the OS X environment. Cachebuster showed a speedup of approximately 65 percent with OBS-L. The SPEC workloads also benefited. The performance of mcf increased by 5 percent, and swim improved approximately 13 percent. As in our Linux experiments, the lighter tasks eon and sixtrack showed slight (2 to 3 percent) degradation. Overall system speedup was just over 3 percent.

These results, although not always as good as our Linux results, are nonetheless encouraging. We observed some unexplained behavior in OS X's scheduling of tasks and interrupt handlers, which may have diminished our system's benefits. We plan to continue experimentation with newer public versions of OS X as they become available.

Policy: Migrating across caches

OBS-L uses observations to affect scheduler decisions. We created an additional policy engine called OBS-X (for cross-cache), which uses observations to affect task migration decisions in the OS. OBS-X's goal is to distribute cache-heavy threads throughout the system, not only helping spread out cache load, but also providing more opportunity for OBS-L to achieve benefits with its local policies.

In Linux, task migration occurs as part of the load balancer. The default policy attempts to keep all run-queues the same length, while minimizing migration costs. The OBS-X prototype uses its observations of each task's cache usage to alter the load balancer's decisions.

When a new task is created, OBS-X looks for the LLC group with the smallest cache load. The new task is placed in this group and then becomes eligible for migration through any of the policies in force.

OBS-X also includes the notion of overweight tasks. For each LLC group, OBS-X tracks the average cache weight of the tasks in that group. A task is overweight if its cache weight is greater than the average weight in its group. OBS-X attempts to prevent migration of overweight tasks, both to avoid excessive migration costs (such tasks, if migrated, have to warm up a new cache) and to prevent these tasks from creating new cache interference in a new location.

Periodically, OBS-X balances the cache load by moving an overweight task from the cache-heaviest LLC group into the currently lightest LLC group. Despite the potential migration cost, this lets the system dynamically adjust and seek optimal distribution of cache-intensive tasks as the overall workload changes over time.

Results

We ran two sets of experiments across four cores in two LLC groups, using the

IEEE Micro micr-28-03-knau.3d 28/5/08 14:45:17 58 Cust # Knauerhase

same job mix as in the previous OBS-L experiments. Because the number of cores was doubled, the number of threads also doubled for each workload. We ran each experiment continuously for 30 minutes, restarting each task until the period completed.

The first set of experiments consisted of four instances of cachebuster and four instances of spinloop, totaling eight threads on four cores. For this workload, OBS-L with the default Linux load balancer produced between 8 and 18 percent speedups for the heavy cachebuster tasks. With the addition of OBS-X, cachebuster performance increased between 12 percent and 62 percent. The reason for the increase is that OBS-X distributed the cache-heavy tasks across LLC groups, thus minimizing the scheduling of heavy tasks together. To verify this, we traced the migration of cachebuster and spinloop tasks while running OBS-X.

Figure 3 (next page) shows the migration of cachebuster tasks between two LLC groups over a 10-minute interval with OBS-X running. The four cachebuster tasks are indicated in different shades of gray. The spinloop tasks are not shown. Most of the time, OBS-X ensures that the system is balanced; for example, there are only two heavy cachebuster tasks in each LLC group. Occasionally, three cachebuster instances briefly share an LLC before OBS-X makes its corrections.

The second set of experiments used the same SPEC job mix as the OBS-L experiments shown in Figure 2. We ran four instances each of mcf, eon, sixtrack, and swim, totaling 16 threads on four cores. In terms of OBS weight, mcf is the heaviest, swim is medium, and eon and sixtrack are the lightest tasks. Figure 4 (next page) shows the results for our comparison workload. For simplicity, we show aggregated results for each task (the average of results for each instantiation). The ''stock'' group indicates performance under unmodified Linux; we normalize to this value, so each bar shows 1.0. The OBS-L group shows an overall speedup of 3.2 percent with four cores. Adding OBS-X, the overall speedup increases to 4.6 percent. As in Figure 3, the heavy tasks are evenly distributed most of the time, but there are short intervals when they are not. In Figure 4, the heavier tasks (mcf and swim) improve from stock to OBS-L to OBS-X; however, the lighter tasks show less than 3 percent degradation.

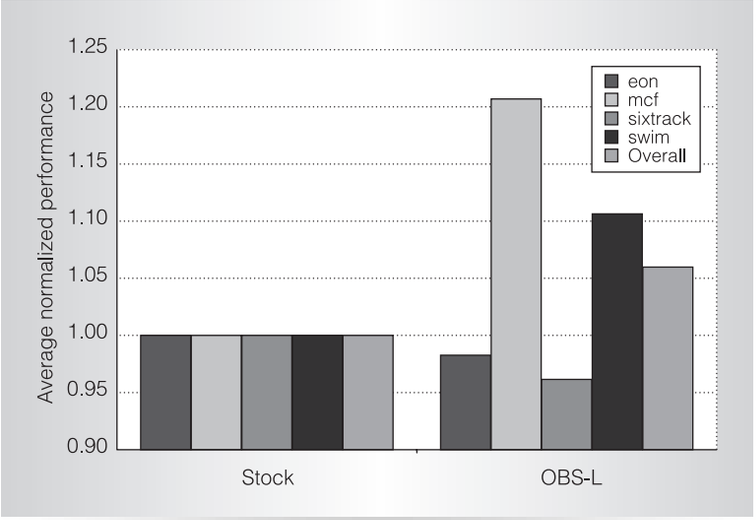

Figure 2. OBS-L performance benefit for SPEC workload on Linux (stock: unmodified Linux).

Policy: Addressing fairness

In experimenting with OBS-L and OBS-X, we found that cache-light tasks could experience a small performance degradation (less than 5 percent) because they were more likely to be coscheduled with cache-heavy tasks. These changes show that although we maintain fair access to CPU time based on priority, fair access to the CPU is insufficient to ensure that all applications make equal progress. To alleviate this, we added an extra policy heuristic to credit CPU time back to applications that suffer from running with heavy tasks. Under this policy (implemented as OBS-C), the system computes the difference in weights between any coscheduled tasks and transfers CPU time (t_{credit}) proportionally from the heavier to the lighter task. As the following equation shows, the amount of credit is proportional to the weight difference between the tasks, scaled by the amount of time the light task overlapped, and multiplied by a constant c_{credit} representing the percentage of system time reserved for

IEEE Micro micr-28-03-knau.3d 28/5/08 14:45:17 59 Cust # Knauerhase

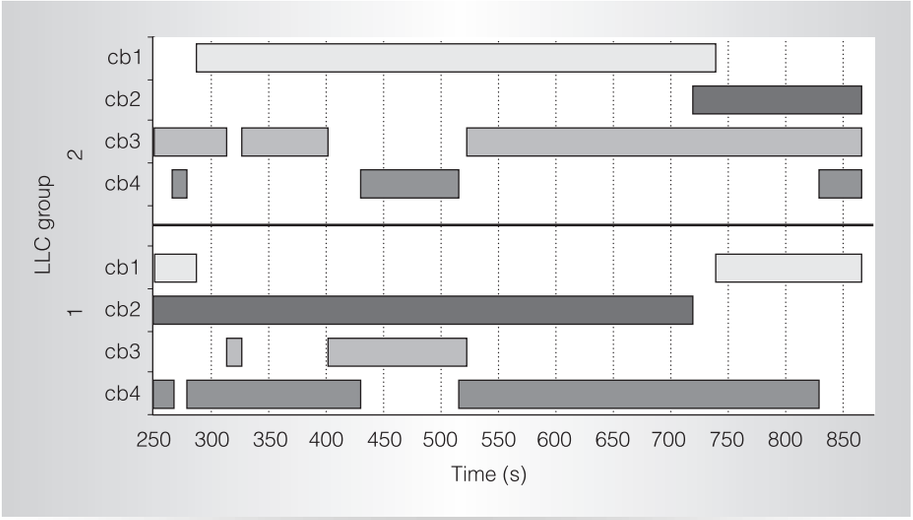

Figure 3. Cachebuster (cb) task migration with OBS-X.

allocation to tasks that suffer degradation:

$$t_{\text{credit}} = \frac{w_{\text{light\_task}}}{w_{\text{largest}} - w_{\text{smallest}}} \times t_{\text{overlap}} \times c_{\text{credit}}$$

Like the policy proposed by Fedorova, Seltzer, and Smith,[9] OBS-C adjusts CPU time to compensate for differences in cache utilization. In contrast, however, OBS-C's use of the task's cache weight eliminates the training phase required to estimate the uncontended fair cache miss rate.

Results

For our OBS-C policy, we evaluated values of c_{credit} between 0 and 5 percent in experiments with seven varied workloads composed from 13 SPEC CPU 2000 benchmarks. Figure 5 illustrates results using the same job mix from our "reducing cache interference" prototype, described earlier as OBS-L, showing the range of performance variation versus stock Linux for all applications. From these experiments, we heuristically determined that a c_{credit} value of 2 percent achieved the preferred trade-off between the policy's goal—that no

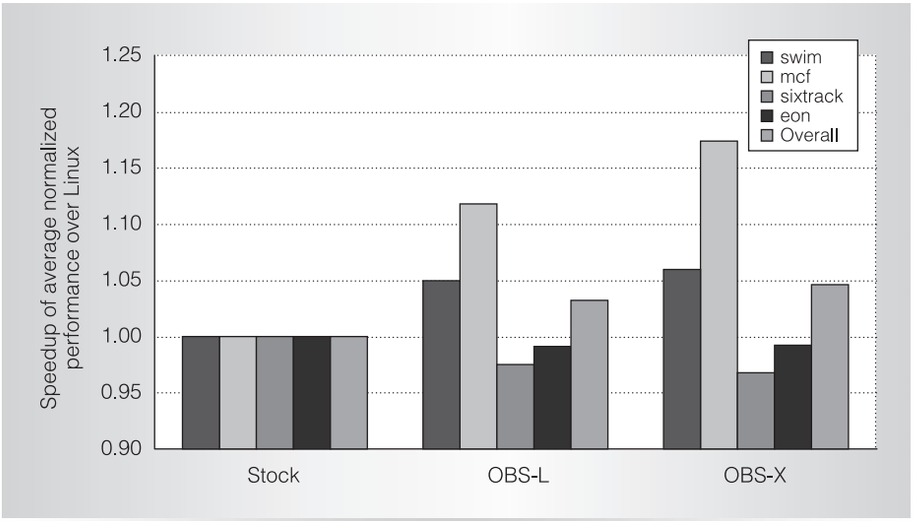

Figure 4. OBS-X performance benefit (16 threads and four cores).

IEEE Micro micr-28-03-knau.3d 28/5/08 14:45:20 60 Cust # Knauerhase

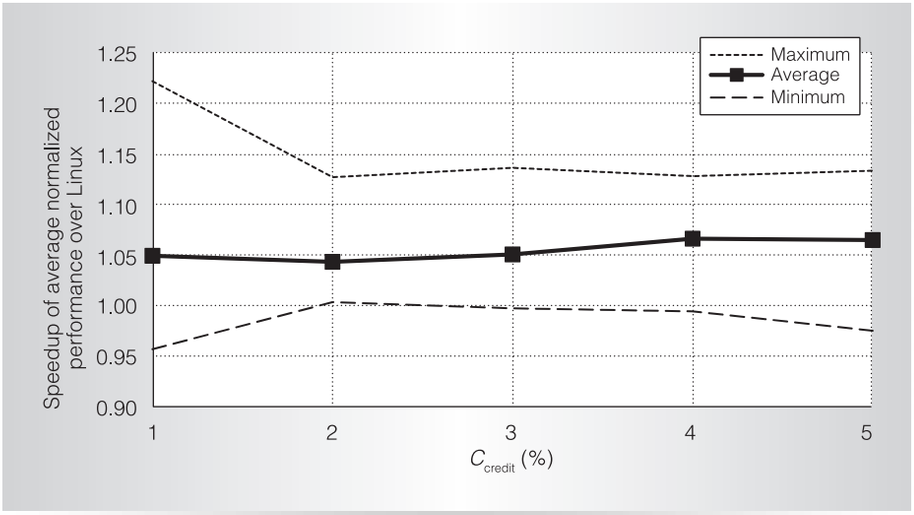

Figure 5. OBS-C performance benefits for various values of c_{credit}. Max is value of the speedup for the task that showed the greatest improvement. Min is the speedup (slowdown if <1.0) of the task that showed the least improvement. Average is the overall system benefit.

tasks perform worse under OBS-C than under stock Linux—and maximizing overall performance.

Figure 6 (next page) shows that the addition of OBS-C to OBS-X almost completely eliminated performance degradation (the worst case is 99.7 percent of stock Linux performance), while the average speedup increased from 4.6 to 5.6 percent.

Policy: Accommodating functional asymmetry

Using another part of OBS-M's observation data—the frequency of certain instruction types—we built a policy engine (OBS-F) to accommodate processors that offer different features on different cores.

Lacking real asymmetric hardware, we designed an infrastructure to simulate functional asymmetry in an existing multicore system. We used CPU-specific flags to disable floating-point (FP) and SIMD instructions (including MMX and streaming SIMD) instructions on a subset of cores in the system. With these functions disabled, invalid instructions executing on a core trigger a device-not-available (DNA) exception. Linux itself uses a similar mechanism to save FP state when a thread's context ends; however, Linux resets the CPU flags to permit FP operations after such a fault.

The most basic part of OBS-F's policy is to observe the DNA fault that arrives and store this information among its observations of the task. Rather than simply letting the application fail, OBS-F catches the fault and forces the task to migrate to a core that can execute the faulting instruction. We call this naive policy OBS-F1.

A second version of the policy (OBS-F2) monitors the task after it migrates, looking for a sufficiently long period of not issuing instructions that would fault on a non-fully-featured core. After a sufficiently long period, it re-enables the task to migrate freely, and the other OBS policies (or the OS default policies) can move the task through the system.

In another aspect of the policy, OBS-F tracks the accumulation of unavailable instructions over time (total number and number of faulting quanta). Using the task's history, OBS-F can determine that the frequency of faulting instructions is high enough that the task should be banned from a core for the rest of its life in the system, saving both migration costs and cache interference caused by a task's frequently moving to and from FP-enabled cores.

IEEE Micro micr-28-03-knau.3d 28/5/08 14:45:29 61 Cust # Knauerhase

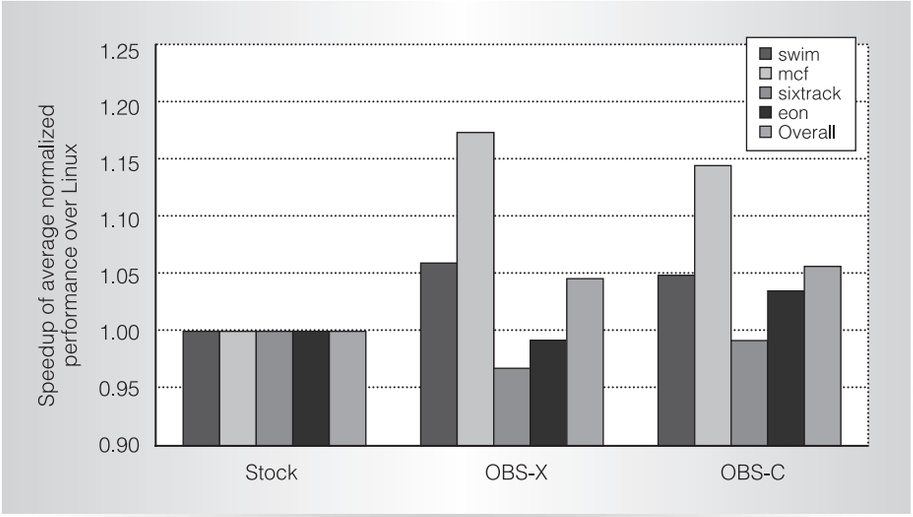

Figure 6. OBS-C performance benefit (16 threads and four cores).

To explore the effects of these policies, we parameterized the OBS-F prototype to allow experimentation with various settings. Most important among these are thresholds on the number and frequency of invalid operations and the length of time without invalid operations. Depending on the cost of the DNA fault, the cost of migrating a task among cores, and the number of fully featured versus partially disabled cores, other thresholds in the OBS-F policy engine might be more advantageous to a particular platform.

Results

We ran various workloads on our scheduler, using our quad-core test system. Cores 1 and 3 are little: that is, we disabled the special instructions FP and SIMD on them. Cores 0 and 2 are big cores—that is, they have a complete feature set. We used a workload consisting of four programs from SPEC CPU 2000:

- gzip, which periodically uses special instructions;
- vortex, which uses special instructions only at the beginning and end of its execution; and
- facerec and sixtrack, which heavily use special instructions throughout their execution.

Figure 7 shows the special-instruction count we measured over a representative period for gzip and facerec.

We ran this workload in three configurations. Figure 8 shows the results. The first and most basic configuration is stock Linux. Since no task in our workload is entirely without special instructions, each one crashes when run on a little core; we show this in the graph with a performance of 0 for stock. OBS-F1 observes the fault and moves offending tasks to big cores, disallowing migration to a core without the required features. The figure shows our normalization of each task's performance to 1.

Our OBS-F2 policy allowed migration back to little cores. Our heuristic was to re-enable migration if the task didn't perform any special instructions in the previous quantum. We found this a reasonable heuristic given the workloads and migration costs of this platform. The results were a greater performance (doubling or nearly doubling) because our observations and policies allowed an effective assignment of each task according to its behavior.

Other observation-driven policies

There are several other policies that can be easily implemented with our observation-based scheduling methods. We have

IEEE Micro micr-28-03-knau.3d 28/5/08 14:45:32 62 Cust # Knauerhase

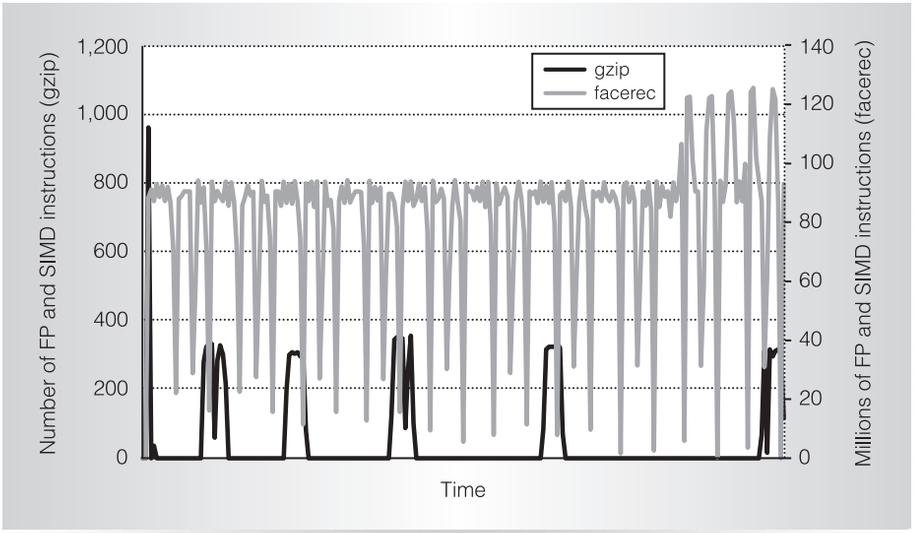

Figure 7. Instruction use of two sample workloads.

begun investigation of several of these, specifically looking at contention for in-core functional units, different power states among cores, and applicability of our work to virtualized multicore environments.

Reducing functional-unit interference

Hyperthreaded processors present the illusion of two cores by implementing two instruction pipelines that share functional units in one processor or one core. Properly implemented hyperthreading improves performance by allowing parallel execution in a core. Increased transistor densities allow the duplication of more functional units in a core; however, it is not clear that all cores in a system will have the same set of duplicated features. Nakajima and Pallipadi describe preliminary techniques using performance counters to observe memory contention among logical cores.[10]

Hardware observation of contended resources will be an important feature in future processors. Observation of contention, such as we already have implemented for cache and floating point, will allow the OS to implement analogous policies—either to migrate a task somewhere with less contention or to credit CPU time back to a task that has suffered disproportionately from contention.

Multicore power management

Currently available multicore systems support power management, specifically dynamic voltage and frequency scaling (DVFS). In some cases, all cores must run in lockstep at the same frequency. Other implementations allow cores to change power states independently. Initial research indicates that OS-level observation—of both hardware power events and per-task activity—will enable better power management policies.[11] For example, tasks that are largely memory bound might run on a lower-power core and obtain similar per-

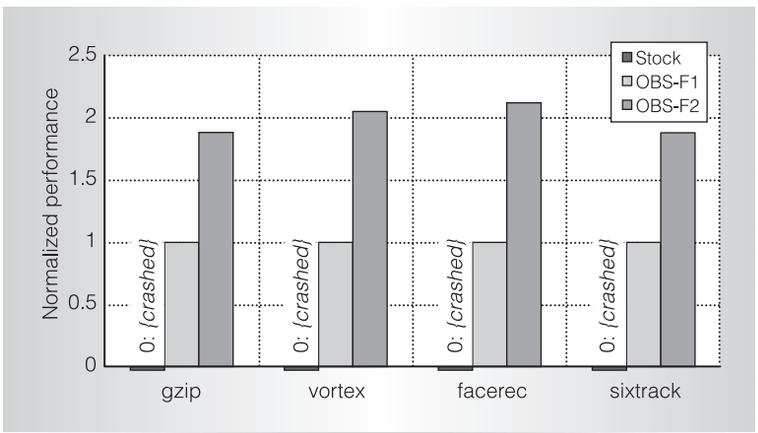

Figure 8. Comparison of OBS-F policies (SPEC CPU 2000 workload).

IEEE Micro micr-28-03-knau.3d 28/5/08 14:45:36 63 Cust # Knauerhase

Related Work

Tam, Azimi, and Stumm implemented a system that, like ours, uses hardware features (Power5-architecture performance-monitoring units) to decide to migrate threads throughout an SMT/CMP/SMP platform.[1] Specifically, the system maintains vectors of cache accesses to detect threads that share data and examines memory access stalls to determine which threads are using data from a faraway cache. The system uses this information to group related threads and attempts to coschedule those groups to maximize sharing and locate threads close to data needed by the threads. Experiments with server workloads demonstrated a 5 to 8 percent overall speedup in each case.

Suh, Devadas, and Rudolph propose a different methodology,[2] which defines *marginal gain* as the derivative of a task's cache-miss ratio curve over time.[2] To measure marginal gain, they propose a set of hardware counters and modifications to cache controllers. They use this metric for task scheduling task and cache partitioning. In simulations of various cache sizes and configurations, the authors show that their observations produce significantly fewer cache misses for several SPEC applications.

In contrast to our focus on throughput, Kim, Chandra, and Solihin have explored methods to ensure fairness—specifically, making sure the effect of cache contention is uniform across tasks.[3] They use a stack-distance profile[4] (similar to marginal gain[2]) as input to two schemes. In a static scheme, they run each candidate task alone to get metrics used to partition the cache when those tasks are later coscheduled. To accommodate varied runtime task behavior, a dynamic scheme alters cache partitions periodically using performance deltas from the immediately previous repartitioning. Both schemes require either hardware partitioning support or modifications to the cache replacement algorithm. In simulated experiments, the authors succeed at improving fairness and thus derive a nontrivial (8 to 15 percent) performance boost in their simulated hardware.

Cho and Jin share our belief in active OS participation in cache management.[5] Their work involves simulation of a tiled CMP processor, comparing private and shared caches among cores in a platform with a novel architecture in which memory pages map into ''cache slices.'' With this enhancement, the OS, using its knowledge of memory page usage, can make intelligent cache management decisions. The authors demonstrate that moving this functionality into the OS allows flexibility to imitate existing hardware mappings and to provide a virtual multicore processor by placing thread data in similarly shared cache slices and carefully choosing distance from cores to cache slices. Cho and Jin's work would likely complement our research, but reconciling their simulated architecture with our experimental setup would pose a significant challenge. Lin et al. have also done work in which the OS is an active participant; however, their goal is more focused on partitioning the cache among tasks.[6]

Similar to our functionally asymmetric prototype, the MISP (multiple instruction stream processor) system provides fully featured cores, managed by the OS, and small cores, managed by applications via a shred library.[7] The system also adds a user-level hardware fault exception. When an application-managed core incurs an exception or performs a system call, it triggers a user-level fault. The system fault handler (in ring 3, or application space) then migrates the faulting shred's state to one of the OS-managed cores via a new instruction called Signal. This instruction carries certain state information and works like a user-level interprocessor interrupt. Upon receiving the signal, a big core resumes the migrated shred's execution, thus repeating the original fault, which is then handled on the big core in ring 0. After this proxy execution, the OS-managed core migrates the shred back to its original application-managed (small) core and resumes execution.

Wang et al. extend MISP to heterogeneous systems with completely different instruction set architectures in their Exochi system by defining compiler extensions to support user programming in a diverse environment.[8] Their prototype demonstrated an Intel Core 2 processor and an Intel graphics card, with tasks faulting and migrating back and forth for features only offered by the big core.

References

1. D. Tam, R. Azimi, and M. Stumm, ''Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors,'' *Proc. 2nd ACM SIGOPS European Conf. Computer Systems* (EuroSys 07), ACM Press, 2007, pp. 47-58.
2. G.E. Suh, S. Devadas, and L. Rudolph, ''A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning,'' *Proc. 8th Int'l Symp. High-Performance Computer Architecture* (HPCA 02), IEEE CS Press, 2002, pp. 117-130.
3. S. Kim, D. Chandra, and Y. Solihin, ''Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture,'' *Proc. 13th Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 04), IEEE CS Press, 2004, pp. 111-122.
4. R.L. Matson et al., ''Evaluation Techniques for Storage Hierarchies,'' *IBM Systems J.*, vol. 9, no. 2, 1970, pp. 78-117.
5. S. Cho and L. Jin, ''Managing Distributed, Shared L2 Caches through OS-Level Page Allocation,'' *Proc. 39th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro 06), IEEE CS Press, 2006, pp. 455-468.
6. J. Lin et al., ''Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems,'' to be published in *Proc. 14th Int'l Symp. High-Performance Computer Architecture* (HPCA 08), IEEE CS Press, 2008.
7. R.A. Hankins et al., ''Multiple Instruction Stream Processor,'' *Proc. 33rd Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 114-127.
8. P.H. Wang et al., ''EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-Core Multithreaded System,'' *Proc. Conf. Programming Language Design and Implementation* (PLDI 07), ACM Press, 2007, pp. 156-166.

IEEE Micro micr-28-03-knau.3d 28/5/08 14:45:43 64 Cust # Knauerhase

formance, whereas computation-intensive tasks might benefit from migration to a higher-speed or higher-power core. We expect that combining our existing per-task observation framework with heuristics relating performance to frequency will improve task scheduling in a DVFS multicore environment.

Virtualization

The main focus of the prototypes described in this article has been to seek optimal mapping of OS tasks to the system's computing resources. We have also investigated observations and policies for mapping virtual machines (VMs) to computing resources. We instrumented the Xen hypervisor to collect similar cache and memory observations on a per-VM basis.[12] We found that runtime observation of behavior is even more important for VMs because an even greater opacity prevents knowledge of what a VM is doing or will do next. Indeed, although compiler or developer hints for application behavior are conceivable, the nature of running applications on an OS in a VM further precludes the chance of getting behavior hints to help in resource mapping.

As usage models for virtualization continue to mature, we expect hypervisor observation to be useful not only for avoiding interference, but also for providing accounting and billing and quality of service assurance to VMs in a multicore system and in clusters of multicore systems.

The ''Related Work'' sidebar summarizes research on using OS observations to improve performance in multicore systems.

Multicore systems have become increasingly prevalent and increasingly complex. Although techniques exist for application tuning in a multicore environment, they cannot be optimized for every possible multicore platform configuration on which a user might run them. Moreover, the dynamic nature of workloads that run concurrently along with an application causes runtime effects that influence application performance differently for each platform and workload configuration.

Our research shows that the OS can help this problem by making dynamic observations of task behavior (without requiring application involvement) and then implementing smarter policies based on the results of these observations. We believe there is more fruitful work to be done in this area. MICRO

Acknowledgments

We thank our colleagues Ravi Iyer, Don Newell, Li Zhao, and Jaideep Moses for their significant contributions to the scheduler algorithms used in this article and their detailed analyses of cache behavior of SPEC workloads. We are also grateful for the anonymous reviewers' detailed comments, which helped shape the final version of this article.

References

1. L. Hammond, B.A. Nayfeh, and K. Olukotun, ''A Single-Chip Multiprocessor,'' *Computer*, vol. 30, no. 9, Sept. 1997, pp. 79-85.
2. Intel Corp, ''A New Era of Architectural Innovation Arrives with Intel Dual-Core Processors,'' *Technology@Intel Magazine,* May 2005, http://www.intel.com/technology/magazine/computing/Dual-core-0505.pdf.
3. D.T. Marr et al., ''Hyper-Threading Technology Architecture and Microarchitecture,'' *Intel Technology J.*, vol. 6, no. 1, Feb. 2002, http://www.intel.com/technology/itj/archive/2002.htm.
4. J. Rattner, ''Tera-Scale Computing—A Parallel Path to the Future,'' *Intel Software Network,* Jan. 2005, http://softwarecommunity.intel.com/articles/eng/1275.htm.
5. Intel, *64 and IA-32 Architectures Software Developer's Manual*, May 2007, http://www.intel.com/products/processor/manuals/index.htm.
6. J.R. Bulpin and I.A. Pratt, ''Hyper-Threading Aware Process Scheduling Heuristics,'' *Proc. USENIX Ann. Tech. Conf,* 2005, Advanced Computing Systems Assoc., pp. 103-106.
7. L. Zhao et al., ''CacheScouts: Fine-Grained Monitoring of Shared Caches in CMP Platforms,'' *Proc. 16th Int'l Conf. Parallel Architecture and Compilation Techniques*

(PACT 07), IEEE CS Press, 2007, pp. 339-352.
8. X. Zhang et al., ''Processor Hardware Counter Statistics as a First-Class System Resource,'' *Proc. 11th Workshop on Hot Topics in Operating Systems* 2005, http://www.usenix.org/event/hotos07/tech/full_papers/zhang/zhang_html/hotos07.html.
9. A. Fedorova, M. Seltzer, and M.D. Smith, ''Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler,'' *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques* (PACT 07), 2007.
10. J. Nakajima and V. Pallipadi, ''Enhancements for Hyperthreading Technology in the Operating System: Seeking Optimal Scheduling,'' *Proc. 2nd USENIX Workshop Industrial Experiences with Systems Software* (WIESS 02), Advanced Computing Systems Assoc., 2002, pp. 35-38.
11. Y. Koh et al., ''An Analysis of Performance Interference Effects in Virtual Environments,'' *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software* (ISPASS 07), IEEE CS Press, 2007, pp. 200-209.
12. D. Tam, R. Azimi, and M. Stumm, ''Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors,'' *Proc. 2nd ACM SIGOPS European Conf. Computer Systems* (EuroSys 07), ACM Press, 2007, pp. 47-58.

Rob Knauerhase is a staff research scientist at Intel Labs. His research interests include system software, machine virtualization, mobile computing and communications, and information privacy in the digital world. Knauerhase has an MS in computer science from the University of Illinois at Urbana-Champaign. He is a senior member of the IEEE.

Paul Brett is a senior software engineer at Intel Labs. His professional interests include virtualization, distributed systems, and system software. Brett has a first-class honours degree in systems engineering from the Open University, UK.

Barbara Hohlt is a research scientist at Intel Labs. Her research interests include operating systems, distributed systems, and networks. Hohlt has a PhD in computer science from the University of California at Berkeley.

Tong Li is a senior software engineer at Intel Labs. His research interests include processor microarchitecture and operating systems. Li has a PhD in computer science from Duke University.

Scott Hahn is a principal engineer at Intel Labs, where he leads the Operating System Research Group. His professional interests include operating systems, data networking, supercomputer networks, and network management. Hahn has an MS in computer science from Union College.

Direct questions and comments about this article to Rob Knauerhase, Intel Labs, Intel Corp., 2111 NE 25th Ave., Mailstop JF2-55, Hillsboro, OR 97124-5961; knauer@ichips.intel.com.

IEEE Micro micr-28-03-knau.3d 28/5/08 14:45:49 66 Cust # Knauerhase